ABSTRACT
For many business users, the format procedure might be their favourite SAS procedure. The procedure takes the confusing coded data values stored on databases and presents a readable report. For the SAS programmer however, the coded data or format catalogs can be confusing. This is often the case with program code inherited from other people, especially when the documentation is poor. But the format procedure is very powerful, and a lot of its power is not used by programmers because they haven’t the time to “make the report pretty”, when they consider their job is to “analyse the data”.

In fact, formatting of data is not just a “prettying up” exercise for the business user, formatted data can very quickly tell the analyst of problems in the data. Formats can also be used to provide common tools to a team of programmers, or a suite of programs. This sharing of formats can save a great deal of development, analysis and reporting time.

We’ll start by building some simple formats, and show how these can be shared and implemented more widely. Then we’ll look at building self-modifying formats from our data, and share these between different SAS platforms.

INTRODUCTION
This paper assumes an understanding of simple SAS data steps. No knowledge of creating SAS formats is required.

BUILDING A FORMAT WITH FORMAT PROCEDURE CODE.

Most of us use the format procedure to display data in a more “user friendly”, or convenient manner. Often we are working with data that has been stored in a coded format. Printing statistics based on coded data produces information that might be hard to understand. Consider the following data step.

```sas
47    Data _NULL_;  
48      Set ACCOUNTS;  
49      By STATUS;  
50      If First.STATUS  Then COUNT = 1;  
51      Else COUNT ++ 1;  
52      If Last.STATUS  Then Put  STATUS = COUNT =;  
53    Run;  
```

STATUS=A COUNT=2473  
STATUS=B COUNT=855  
STATUS=C COUNT=1636  
STATUS=D COUNT=2507  
STATUS=E COUNT=812  
STATUS=X COUNT=1717  

NOTE: There were 10000 observations read from the data set WORK.ACCOUNTS.  
NOTE: DATA statement used:  
real time 0.02 seconds  
cpu time 0.02 seconds  

To get this result, we created a format that associated each value of status with a piece of descriptive text. The following code creates the format we used above.

```sas
9     Proc Format;  
10      Value $FAcStat 'A' = 'Authorised'  
11                     'B' = 'to be authorised'  
12                     'C' = 'Cancelled'  
13                     'D' = 'Denied'  
14                     'E' = 'Pending review'  
15                     'X' = 'Administrative record';  
16    Run;  
```

Within this Procedure call, we have defined an association between two lists of data. One list contains the data we find in our STATUS table. Each of the entries in this list is associated with descriptive text we select from our second list. This association is what we call a “value format”, because we are replacing our original data with new text based on the value of our source data.

```sas
47    Data _NULL_;  
48      Set ACCOUNTS;  
49      By STATUS;  
50      If First.STATUS  Then COUNT = 1;  
51      Else COUNT ++ 1;  
52      If Last.STATUS  Then Put  STATUS = $FAcStat. COUNT =;  
53    Run;  
```

STATUS=Authorised COUNT=2473  
STATUS=to be authorised COUNT=855  
STATUS=Cancelled COUNT=1636  
STATUS=Denied COUNT=2507  
STATUS=Pending review COUNT=812  
STATUS=Administrative record COUNT=1717  

NOTE: There were 10000 observations read from the data set WORK.ACCOUNTS.  
NOTE: DATA statement used:  
real time 0.78 seconds  
cpu time 0.03 seconds  

In this data step we are testing the frequency of accounts with each status. The information in this frequency table would be more useful if we reported a more meaningful value for the status, rather than the obscure status values. The addition of a format to our “Put” statement gives us information we can more readily understand. Notice the change, highlighted in red, and the effect it has on our output.
Let's look briefly at how we can make our format a little more effective in dealing with our data. Suppose that there were a handful of bad records in our data that we wanted to clean up, or report separately. Here is an example of that sort of bad data from our earlier example.

OTHER VALUES IN OUR FORMATS

Suppose that our corporate system should only ever have the values 'A', 'B', 'C', 'D', 'E' and 'X'. To cater for these, we can instruct our format procedure that a certain range of values is acceptable, and any other value is not. The 'Other' option on our VALUE statement will deal with any value we have not specified. Here is how we might code for these and other incorrect values.

The lower case values 'f', 'm', 'r' and 's' are confusing the report we have provided. Suppose that our corporate system should only ever have the values 'A', 'B', 'C', 'D', 'E' and 'X'. To cater for these, we can instruct our format procedure that a certain range of values is acceptable, and any other value is not. The 'Other' option on our VALUE statement will deal with any value we have not specified. Here is how we might code for these and other incorrect values.

The format FacStatted1 might be a different format we wanted to create for a special purpose. This format was not created with the name we provided, no warning or error was issued, and we have now overwritten our original format. The lessons to learn from these examples are that format names, (including the string symbol $ for character formats) must be no longer than 8 bytes. Secondly, we should develop a practice of always carefully reading the log at the end of our jobs, and not concentrate on just WARNING and ERROR messages. Incidentally, some SAS programmers run their code with Notes suppressed through the NoNotes Option. You can see that this would have presented a problem for this example.

Note that the SAS System truncates the name to 8 bytes when it creates the format. Observe too that SAS reports the long name with a NOTE, and that the format is still created without a higher priority message like a WARNING or ERROR. The lack of such a message is a potential problem, and is a reminder that the SAS log should be carefully read. The truncation is possibly a more serious problem when we look at what happens when we try to create a second different format, and mistakenly provide a name more than 8 bytes in length.

The lower case values 'f', 'm', 'r' and 's' are confusing the report we have provided. Suppose that our corporate system should only ever have the values 'A', 'B', 'C', 'D', 'E' and 'X'. To cater for these, we can instruct our format procedure that a certain range of values is acceptable, and any other value is not. The 'Other' option on our VALUE statement will deal with any value we have not specified. Here is how we might code for these and other incorrect values.

Let's look briefly at how we can make our format a little more effective in dealing with our data. Suppose that there were a handful of bad records in our data that we wanted to clean up, or report separately. Here is an example of that sort of bad data from our earlier example.
If we were to use this format in a summarisation procedure such as FREQ, SUMMARY, MEANS, UNIVARIATE, TABULATE or REPORT then the incorrect values would all be grouped into one level of the summary. Here we see the value of the format in highlighting problems with the data. It has clearly indicated to the analyst that certain values exist which are not known to the format table, and are incorrect.

NUMERIC FORMATS

So far we have concerned ourselves with character formats, and while these are probably the more useful since they make reading coded values easier, their numeric cousins can be just as useful. While most of us have used the "Commaw.d", "Datew." or "Dynamicw.d" formats without a second thought, there is much more that can be done with numeric formats.

Consider the problem of reporting a wide range of currency values. Our concern might be how many payments are made in each of four bands, and how many fall outside the range of payments we think normal. So we'll divide the currency range into four bands and build a format as follows.

257 Proc Format;
258 Value FHowMuch 0 - 10 = 'Under £10'
259 10 - 250 = 'Under £250'
260 250 - 2000 = 'Under £2000'
261 2000 - 5000 = 'Under £5000';
NOTE: Format FHOWMUCH has been output.
262 Run;
263
NOTE: PROCEDURE FORMAT used:
real time 0.00 seconds
cpu time 0.00 seconds

This is a simple format, and we can expect it will group our payments and give us some rough analysis of the distribution of values. When we use the format in a call to the Freq Procedure however, we discover our data is not as we expected. Here is the code used as well as an excerpt from the procedure output.

55 Proc Freq Data = PAYMENTS;
56 Tables PAID;
57 Format PAID FHowMuch.;
58 Run;

NOTE: There were 50000 observations read from the data set WORK.PAYMENTS.
NOTE: PROCEDURE FREQ used:
real time 0.34 seconds
cpu time 0.16 seconds

PAID Frequency Percent
-0.03024382 1 0.00 95
Under £10 97 0.19 192
Under £250 2461 4.92 2653
Under £2000 17335 34.67 19988
Under £5000 29915 59.83 49903
5000.147499 1 0.00 49904
5000.227616 1 0.00 49905

You'll notice from the fourth column "Cumulative Frequency" that the value of -0.03024382 is the 95th in the table. The last value shown is also not the last in our data set, and with 50,000 payments in the original file, you'll realise our output produced our 4 neat summaries as expected, as well as 190 or so additional lines from values not covered in the format.

We could use our 'OTHER' keyword, as we did with the character format. However, it might be more valuable to split our other values into two groups. We need to modify our code to tell the Freq procedure that any value lower than our smallest expected value should be in one group, and any value higher than our largest expected value should be in another group. We do this with the keywords LOW and HIGH as in the following code, and resultant output.

291 Proc Format;
292 Value FHowMuch LOW = '*** Is this a credit? ***'
293 0 - 10 = 'Under £10'
294 10 - 250 = 'Under £250'
295 250 - 2000 = 'Under £2000'
296 2000 - 5000 = 'Under £5000'
297 5000 - HIGH = '** Over £5000**';
NOTE: Format FHOWMUCH has been output.
298 Run;

Note that we can use the keyword LOW before any other range declaration, and the SAS compiler will accept it. However, if we try to specify HIGH on its own, we will get an error. This means that we need to specify a lower bound for the 'High value' range, as we have done here.

We might ask, is £5000 in the 'Under £5000' category, or in the 'Over £5000' category. (The accountants among us will insist on knowing, and argue that it should be neither.) We can test for this with a simple data step.

303 Data _NULL_;
304 PAID = 5000;
305 Put PAID= FHowMuch.;
306 Run;
PAID=Under £5000

NOTE: DATA statement used:
real time 0.75 seconds
cpu time 0.01 seconds

It's clear that the range we labelled "Under £5000" includes the value £5000, which would be misleading. It wasn't clear from our code whether the '5000' value should be in the first, or the second assignment. We can address this with a simple change to the format.

The addition of a "less than" symbol '<' will instruct the SAS System to compile the format with all values up to, but not including the upper value in a range. This way, we needn't specify 4999.9999 as our upper bound. Here is the changed format, and our data test again.

322 Proc Format;
323 Value FHowMuch Low = '*** Is this a
The benefit of including the '<' symbol is that the format code is self-explanatory. In fact, the default behaviour of SAS is that it will order the format statements in ascending value, and then assign the upper bound of the range to the value on the right hand side of the range specification unless the symbol '<' specifies otherwise. You may have thought that the order of the format statements provided the order, but this isn't so, as the following code demonstrates. Note that we have specified our 'Under £5000' assignment after the upper range, yet the interpretation of the £5000 value is still the same as in the earlier code.

Let's look now at the other major group of formats. Our VALUE formats are useful for grouping data values into distinct classes or groups. However, sometimes we want to see the actual value, but have it displayed in a particular way. SAS own formats 'Commaw.d', 'Dollarw.d', 'w.d', 'NegParenw.d', 'Percentw.d', and 'SSNw.' are examples of values represented without any grouping, but with a particular appearance.

Let's start by looking at a common type of summary for our payments data. We'll begin by creating a copy of the PAID column, calling it PAIDVAL. This will allow us to perform both a CLASS analysis on the payment using our FHowMuch format, and a value analysis on the total amount.

And here at figure 1 is the table we created.

All along we have been talking about payments in pounds. On the face of this table however, there is nothing to indicate which
currency we are using. What we need is a format that will add
the appropriate currency symbol, and retain the formatting we
can see from SAS' own Commaw.d format.

Since what we are creating is not changing the value, but is
changing the appearance of the value instead, we use what we
call a 'Picture' format. Here is how we might create a format for
pounds sterling, and at figure 2 is the result of using that format in
the table we have just seen.

```
188 Proc Format;
189     Picture GBPound Low -< 0   =
190       '000,000,000,000,000.00' (prefix = '-£')
191           0   - HIGH =
192       '000,000,000,000,000.00' (prefix = '£');
193 NOTE: Format GBPOUND has been output.
194 Run;
```

NOTE: PROCEDURE FORMAT used:
real time 0.74 seconds
cpu time 0.01 seconds

99 Proc Tabulate Data = PAYMENTS
Missing;
100 Class PAID;
101 Var PAIDVAL;
102 Table PAID = 'Payment band' ALL =
103       'All groups',
104           N = 'No. of payments' * F =
105       'Comma12.
106           PctN = '% in band' * F = 6.3
107           PAIDVAL = ' '  * Sum = 'Total
108           value' * F = GBPound18.2 /
109           Rts = 25  Box = 'Banded payment
110           analysis';
111 Format PAID FHowMuch.;
112 Run;

NOTE: There were 50000 observations read from
the data set WORK.PAYMENTS.
NOTE: PROCEDURE TABULATE used:
real time 0.15 seconds
cpu time 0.15 seconds

Look at the code we used to create the format. We have used
our LOW and HIGH keywords, to ensure all possible values are
covered. We have also specified a prefix for each format that is
then written at the beginning of the output value.

When we used the format, we specified a length and number of
decimal places for the output (GBPound18.2). This is a very
useful trick to remember, since it means we can specify quite
long strings in a standard format, perhaps allowing for billions of
pounds, and then only display the width we want.

While we're looking at picture formats, let's deal with one of the
issues I have with the Tabulate Procedure. Notice that the
column for PctN, which should be displaying a percentage, is
displaying numbers that sum to 100 in the 'All groups' row. If you
try to use the Percentw.d format, the percentages displayed will
be incorrect. That's because the Percentw.d format expects the
percentages to be less than 1. We can solve this with a picture
format in the following way.

```
228 Proc Format;
229     Picture PctTab LOW -< 0   = '
230       009.00%' (prefix = '-')
231                     0   - HIGH =
232       '009.00%';
233 NOTE: Format PCTTAB has been output.
234 Run;
```

NOTE: PROCEDURE FORMAT used:
real time 0.01 seconds
cpu time 0.01 seconds

Here at figure 3 is the output, using the new percentage format.

Just before we leave manual creation of formats, there is
something worth pointing out. Notice that the percentage for
"** Over 5000 **" is displayed with a leading zero before the decimal
point, but the currency format does not have a leading zero after
the '£' symbol. The reason for this is in the Picture format code.
Where a 0 is used as a placeholder in the format picture
"'000,000,000,000,000.00' (prefix = '-£')", if the leading value is 0,
then the 0 will not be displayed. Replacing the '0' with a '9'
causes the 0 digits to be displayed "'009.00%' (prefix = '-')."

MANAGING THE FORMAT CATALOG

Now the sharp-eyed reader may have noticed something a little
different about the recreation of the character format above. The
first time we recreated the format, when we had an incorrect
format name, the following message appeared in the log.

```
NOTE: Format $FACSTAT is already on the
      library.
```

It wasn't there the second time, and the reason it was missing
was because I deleted the entry first. Understanding the nature of
format entries is useful to us when we create our own formats,
so we’ll briefly explore the nature of a format.

When you execute the format code, a catalog called FORMATS is updated. A new or changed entry is made in the catalog for the format you are creating or modifying. If you browse the catalog, you will see the formats you create as individual entries. Here at figure 4 is a screenshot from my formats catalog, showing some of the entries I have created.

Figure 4

The entries in a format catalog are created with a type of either FORMAT for numeric formats, or FORMATC for character formats. If we have update access to the catalog, we can easily add entries, alter existing entries, rename entries, or delete them. The catalog browser gives us one means of doing this, but we can also delete an entry with code. Here is the code I used to delete the FACStat format before I recreated it.

```
39    Proc Catalog Cat = WORK.FORMATS;
40      Delete FACSTAT.FormatC;
41    Quit;
```

NOTE: Deleting entry FACSTAT_FORMATC in catalog WORK.FORMATS.

Do you remember that I suggested you should read your logs carefully? By deleting existing entries first, I reduce the number of notes around the point where I recreate formats. This makes the log a little easier to read, removing half of the Notes I would get for all entries I was recreating or replacing.

NESTING FORMATS.

Sometimes, we want to display a value normally for all cases except for a handful of exceptions. These exceptions might be values below or above a given amount, or equal to a given amount. Perhaps if our payments file has negative values, we may want to display an exception flag, or if we have payments of 0 value we want to use a special message.

For all other cases we have a format available that is quite adequate, and we don’t want to have to recreate a standard format just to include our special format. To demonstrate this problem, we have a payment file with 30 payments of a standard tuition fee, 6 payments with reimbursements that we want to flag with a special character, and 1 ‘0 value’ payment we want to replace with a special message. Here is a frequency table of the data.

<table>
<thead>
<tr>
<th>PAID</th>
<th>Frequency</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td>6</td>
<td>16.22</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2.70</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>81.08</td>
</tr>
</tbody>
</table>

To take our existing format, add our two special formats, and display our data as we want, here is the code we might run. It is followed by the frequency output it generates.

```
301   Proc Format;
302     Picture FSPECPD LOW <- 0 = '000,000,000.00 **' (Prefix = '-£')
303                      0 = '9.99 ***'
304                  Other = ['GBPound';
305 Run;

NOTE: Format FSPECPD has been output.
```

PAID     Frequency     Percent
---      ------         ----
-£10.00 **         6       16.22
0.00 ***          1        2.70
£30.00           30       81.08

To nest a standard format within a special set of exceptions, we need only add the format name within square brackets ‘[]’ after we define the special cases. While this example is fairly trivial, if we want to capture a specific set of numbers, such as payments of exceptional amounts, it is easier to take this approach than to try to recode the whole format set again.

Sharing format catalogs.

For many programs and programmers, the format procedure is used at the start of the program to create formats for data reported in that particular job. It is more likely however that a piece of data can be analysed and reported in a number of programs. So we may find the same piece of format code at the head of each program. This is a problem, not just for the unnecessary repetition and duplication of the code, but also a maintenance nightmare when we need to change the format code for new code values.

One of the simplest solutions to this problem is to store the format code in a separate program. Then we may include that program into the head of each of our reporting programs. A statement like the following would include a format program.

```
%Include 'D:\Sascode\Production\formats.sas';
```

In some ways, this is a very effective solution. We can:
- Separate the formats into a single program block to maintain Make changes to the format creation code at almost any time.
- Develop and test the code in isolation.
- Include comprehensive and descriptive documentation in comment blocks in the program, which will provide the programmer with a good resource for analysing the data.

Its shortcoming is that larger organisations can have many codetable fields, and some of these can have very many values. The code then becomes quite complex to read and maintain. If for instance we have a field that is reported in one grouping for a business unit, and another for the Finance Division, then we will now have two blocks of format code associated with the same
large block of data. For such a large piece of code, we may now have a significant part of our job time spent in building the format entries, and a large part of our program log filled with notes associated with the format build process.

We can solve some of these problems if we can save the compiled formats, and simply share that compiled block between programs. We still have a program that may have hundreds of thousands of lines of code, and we still have a problem with maintaining the code, but we aren't wasting time and resources by recompiling our formats in each program we run.

As we noted earlier, the formats we created with our included code were written to a catalog in the work library. To share our formats, we need to write them to a permanent format library instead. To do this, we specify the library name in the beginning of our format procedure. Here is the syntax we might use:

```
Proc Format Lib = LIBRARY;
```

The library reference LIBRARY is used by the SAS System as a default location for formats, and if we assign a physical location to this library reference in each program, then the formats will be retrieved from the format catalog. Assigning the reference is done in the same way we assign any reference. We may write it this way:

```
LibName LIBRARY 'D:\Sasdata\Formats';
```

To make sure every job had this library available, we'd probably include this line in our AUTOEXEC program.

Suppose however that we have a format called FAgency in our format catalog, and this FAgency format reported agency codes with long names. For a test program we may want to group together the 15 different branches of ABC Associates into one company name. So we will create and use a format just for this purpose. For convenience, we want to replace the existing FAgency format with a special version for this program only.

If we add the code to our test program to create the format, then we will want this format to be used in preference to our permanent format. To instruct the SAS System to check for temporary formats first, we can set a SAS System option that will assign a number of format catalogs, and define the sequence in which they are checked. The following SAS options statement will force the SAS session to search the WORK catalog first, and then the LIBRARY catalog if the format entry is not found. In this way, the version of FAgency we created temporarily will be used in preference to the permanent version. We needn't modify any other reporting code for the new format, or overwrite the format entry that is shared by all users of the permanent format catalog.

```
Options FmtSearch = ( WORK, LIBRARY);
```

This syntax is not limited to two catalogs. If the Finance Division have specialist formats you want to use, you can include their library reference in the options statement. The following is one way we can achieve this.

```
LibName FINANCE 'D:\Sasdata\Finance';
Options FmtSearch = ( WORK, LIBRARY, FINANCE);
```

In this case, we will search for a format first in our WORK catalog, then in the permanent LIBRARY catalog, and finally in the FINANCE catalog. It is almost as if we had concatenated the three libraries together to build a 'super set' of formats. The difference between this approach and a 'super set' of formats is that the 'super set' can only have one entry with a given name.

The options statement above would allow each of the three catalogs to contain an entry with the same name. Once again, the order of the catalogs in the FmtSearch option controls the order in which the catalogs are searched for an entry with a given name.

Earlier on we discussed the name of the format, and its appearance in the catalog browser. If you look at your own format catalog, you'll find that the description for the entry can be quite esoteric. Here at figure 5 is the sort of information you'll find by default.

```
Figure 5
```

The description column gives us three numbers that suggest the maximum length allowed for the format is 16 bytes. In fact, the maximum length for the format is 40 bytes, and the default and format lengths are both 15 bytes. If you find this sort of description informative, then you can leave it as is. However, I find that if you are going to go to the trouble of creating permanent format catalogs, then you should label them in some way, just as you label your data. This makes it easier for users of your formats to identify their purpose, and their source.

Strangely enough, there is no syntax that will allow you to specify the description for the format in your Format Procedure. This may be why many people don't label their formats. But as we have discussed, a format is held in a catalog as an entry. It may be stating the obvious, but an entry in a catalog is in a different structure to data. For your format entries, we can use the Catalog Procedure to update the description, and provide a meaningful label. We might use code like the following.

```
326   Proc Catalog Cat = Work.Formats;
327     Modify FAgency.Format( Desc = 'Sugi28_tute: Names of agencies by agency code');
328     Modify PctTab.Format( Desc = 'Sugi28_tute: Apply sterling currency format');
329     Modify GbPound.Format( Desc = 'Sugi28_tute: Payments in analysis bands');
330     Modify FSpecPd.Format( Desc = 'Sugi28_tute: Nest formats, analyse payments');
331     Modify FAgency.Format( Desc = 'Sugi28_tute: Status of accounts');
332   Quit;
```

NOTE: Description changed for entry FAgency in catalog WORK.FORMATS.
NOTE: Description changed for entry PGROUP in catalog WORK.FORMATS.
NOTE: Description changed for entry PAGENCY_FORMAT in catalog WORK.FORMATS.
NOTE: Description changed for entry PAGENCY_FORMAT in catalog WORK.FORMATS.
NOTE: Description changed for entry GBGROUP in catalog WORK.FORMATS.
NOTE: Description changed for entry PGROUP in catalog WORK.FORMATS.
We should read the documentation to find out what fields are required for such a table, and I recommend you read that part of the Language guide, where the Format procedure is discussed. At the moment we will do just that, by creating a Format Control table. All we need is a format created from our source data. In a transaction records by the same field. So now we have our data sequenced. We will probably keep CODETBLE.AGENCY sorted, to translate. The second is that the two data sets need to be wasteful, especially if we have a number of similar data columns. They are reprocessing our data to add one column. This seems rather wasteful, especially if we have a number of similar data columns to translate. The second is that the two data sets need to be sequenced. We will probably keep CODETBLE.AGENCY sorted, or indexed by AGY_CODE, but it is unlikely that we selected our transaction records by the same field. So now we have our data being sorted first, which will take time and processing resources, and we will reprocess the data again for each column we want to translate.

Wouldn’t it be better if we could do the following?

```
Data REPORT;
  Merge SELECTION (In = REPORTME)
  CODETBLE.AGENCY (Keep = AGY_CODE)
  AGENT_NM;
  If REPORTME;
Run;
```

This is a perfectly valid method of making our data easy to understand. However, it has two limitations. The first is that we are reprocessing our data to add one column. This seems rather wasteful, especially if we have a number of similar data columns to translate. The second is that the two data sets need to be sequenced. We will probably keep CODETBLE.AGENCY sorted, or indexed by AGY_CODE, but it is unlikely that we selected our transaction records by the same field. So now we have our data being sorted first, which will take time and processing resources, and we will reprocess the data again for each column we want to translate.

Wouldn’t it be better if we could do the following?

```
Data SELECTION;
  Set SELECTION;
  AGY_NAME = Put( AGY_CODE, FagyCde.);
Run;
```

All we need is a format created from our source data. In a moment we will do just that, by creating a Format Control table. We should read the documentation to find out what fields are required for such a table, and I recommend you read that part of the Language guide, where the Format procedure is discussed. In case we forget or can’t lay our hands on the book or the online documentation, the fastest way to check the structure is to create the reverse table and examine the structure SAS creates. We already have a number of formats in our work formats catalog, formats we created above, so it will be easy to recognise the mapping of the values we typed into our Format procedure statements against the table structure. Here is some code to create a control table from an existing catalog.

```
334 Proc Format Lib = WORK
335 CntlOut = CONTROL;
336 Run;
```

Now if we examine that table, we will find some key values. In the following image (figure n), we can see a view of the table, and a number of column headers are circled in red.
This can be very useful as we will explore later.

**WRITING FORMATS FROM DATA.**

All we need to do to create a format from our Agency table then is to define the fields required above. We do need to be careful however. While field names like START, END and LABEL are not very likely to occur in many data sets, a TYPE column is quite common. This is an occasion where good programming practices pay off. We should always use a KEEP option in the set statement in our data steps. In this case, we will avoid possible contention by using the option in a small data set. (Often, we might only bother with these statements when we are processing large tables and want better performance. It is clear that in this case it is a sound programming practice.)

```plaintext
Data CONTROL;
  Set CODETBLE.AGENCY( Keep = AGY_CODE AGENT_NM)
    Rename = ( AGY_CODE = START AGENT_NM = LABEL);
  FMTNAME = 'FAgency';
  TYPE = 'N';
  END = START;
Run;
```

To process this table, we will use the reverse of the format unload process we saw above.

```plaintext
Proc Format Lib = WORK CntlIn = CONTROL;
Run;
```

It seems simple enough, but there are some traps. The first is the data itself. You need to make certain that the values are non overlapping and unique. If for instance our table contained two entries for AGY_CODE 10302, then this is the sort of error we would see.

```plaintext
25    Proc Format Lib = WORK CntlIn = CONTROL;
26      ERROR: This range is repeated, or values overlap: 10302-10302.
27    Run;
```

WARNING: RUN statement ignored due to previous errors. Submit QUIT; to terminate the procedure. 

**NOTE:** Real time used:
real time 0.17 seconds
cpu time 0.01 seconds

Resolving duplicate format values may not be straightforward. It can be hard to identify the correct value programmatically, so you have a number of options. You can leave your code to produce errors when duplicate data is encountered, or you can cleanse and validate the data. I use a process like the following to validate all my data. It is simple, yet provides clear log entries. Note that we are still assuming that the data in our AGENCY codetable is sorted or indexed by AGY_CODE.

```plaintext
Data CONTROL;
  Set CODETBLE.AGENCY( Keep = AGY_CODE AGENT_NM);
  Rename = ( AGY_CODE = START AGENT_NM = LABEL);
  BY START;
  If First.START + Last.START < 2 Then Put
    'PNB: Agency code values are not unique...' START =   LABEL =;
  If Last.START Then Output;
Run;
```

This process will identify duplicates in our log, with a suitable set of notes, but allow our program to finish.

```plaintext
27    Data CONTROL;
28    Set CODETBLE.AGENCY( Keep = AGY_CODE AGENT_NM) 
29      Rename = ( AGY_CODE = START AGENT_NM = LABEL );
30    BY START;
31    FMTNAME = 'FAgency';
32    TYPE = 'N';
33    END = START;
34    If First.START + Last.START < 2 Then Put
35      'PNB: Agency code values are not unique...' START = LABEL =;
36    If Last.START Then Output;
37    Run;
```

PNB: Agency code values are not unique...START=10302 LABEL=TVATBCS
PNB: Agency code values are not unique...START=10302 LABEL=TVATBCSJVKZMYIL

**NOTE:** There were 102 observations read from the data set CODETBLE.AGENCY.
**NOTE:** The data set WORK.CONTROL has 101 observations and 5 variables.

**NOTE:** DATA statement used:
real time 0.77 seconds
cpu time 0.02 seconds

If there is another column on our source data that will assist with removing duplicate entries, then we can use that as well. Here is an example, where the date of effect identifies the most recent assignment, and we are loading the correct value, while still producing some data validation in the log.

```plaintext
225   Data CONTROL( Drop = DATE_EFF);
226     Set CODETBLE.AGENCY( Keep = AGY_CODE AGENT_NM DATE_EFF)
227       Rename = ( AGY_CODE = START AGENT_NM = LABEL );
228     BY START;
229     FMTNAME = 'FAgency';
230     TYPE = 'N';
231     END = START;
232     If First.START + Last.START < 2 Then Put
233       'PNB: Agency code values are not unique...' START = DATE_EFF = Date9. LABEL =;
234     If Last.START Then Output;
235   Run;
```

PNB: Agency code values are not unique...START=10302 DATE_EFF=10FEB2002 LABEL=RLPKHUC
PNB: Agency code values are not unique...START=10302 DATE_EFF=06MAY2003 LABEL=RLPKHUCHTWHDLBLVH

**NOTE:** There were 102 observations read from the data set CODETBLE.AGENCY.
**NOTE:** The data set WORK.CONTROL has 101 observations and 6 variables.

**NOTE:** DATA statement used:
real time 0.02 seconds
cpu time 0.02 seconds

**SHARING FORMATS.**

There are a number of stumbling blocks with sharing formats across platforms.

```plaintext
START = LABEL =;
  If Last.START Then Output;
Run;
```

SUGI 28 Beginning Tutorials
If you are to share the entries, you may wish to move the catalog from one platform to another. You may not be able to simply move the catalog if the version of SAS is different. This is because the structure of catalogs can change between SAS versions, and one of the systems may not have been upgraded to a later version of SAS. You will then require a process that copies the catalog between the platforms to a temporary area, and then uses an import process that recognises the version difference.

Copying the catalog may be an issue, although the SAS/Connect product will allow the transfer to be reasonably seamless. Without SAS/Connect, you can create a transfer file, and use a binary transfer process like FTP to move the transfer file. Then you import the transfer file to a temporary catalog and use a process to copy the entries from one catalog to the other.

In both cases above, we have sent the catalog to a temporary area and imported the entries. There is a good reason for this; the file structure on Windows and Unix means a catalog is a single file. On Windows, it will be part of a larger file that is a complete SAS library. Replacing the file will delete other members of the library. We also import the entries because there may be other entries in the catalog that are specific to the Operating system that we do not wish to change. We would import the entries with a process like the following. Notice that the source catalog has been created under Version 6.

```sas
15 Libname ACATV6 V6 'D:\Saswork\V6\Sasdata\David';
NOTE: Libref ACATV6 was successfully assigned as follows:
    Engine: V6
    Physical Name: D:\Saswork\V6\Sasdata\David
16  Libname MASTCAT 'D:\Saswork\V6\Jobserve';
NOTE: Libref MASTCAT was successfully assigned as follows:
    Engine: V8
    Physical Name: D:\Saswork\V8\Sasdata\David
17  Proc Catalog Cat = ACATV6.FORMATS;
18  Copy Out = MASTCAT.FORMATS;
19  Quit;
```

NOTE: Copying entry PORT.FORMATC from catalog ACATV6.FORMATS to catalog TEMP.FORMATS.
NOTE: Copying entry DELAYF.FORMAT from catalog ACATV6.FORMATS to catalog TEMP.FORMATS.
NOTE: PROCEDURE CATALOG used:
    real time 0.14 seconds
    cpu time 0.01 seconds

In the two cases above, we have sent the catalog to a temporary area and imported the entries. There is a good reason for this; the file structure on Windows and Unix means a catalog is a single file. On Windows, it will be part of a larger file that is a complete SAS library. Replacing the file will delete other members of the library. We also import the entries because there may be other entries in the catalog that are specific to the Operating system that we do not wish to change. We would import the entries with a process like the following. Notice that the source catalog has been created under Version 6.

```sas
15 Libname ACATV6 V6 'D:\Saswork\V6\Sasdata\David';
NOTE: Libref ACATV6 was successfully assigned as follows:
    Engine: V6
    Physical Name: D:\Saswork\V6\Sasdata\David
16  Libname MASTCAT 'D:\Saswork\V6\Jobserve';
NOTE: Libref MASTCAT was successfully assigned as follows:
    Engine: V8
    Physical Name: D:\Saswork\V8\Sasdata\David
17  Proc Catalog Cat = ACATV6.FORMATS;
18  Copy Out = MASTCAT.FORMATS;
19  Quit;
```

Although this works, if we did not have SAS/Connect to Upload the catalog to a version 6 library, we may have performed the following steps.

```sas
156  /* Create a temporary library.*/
157  libname TEMP 'C:\Temp';
NOTE: Libref TEMP was successfully assigned as follows:
158  Engine: V8
159  Physical Name: C:\Temp
160  /* Copy the permanent format catalog to the temporary library */
161  proc catalog cat = ACATV6.FORMATS;
162  copy out = TEMP.FORMATS;
163  quit;
```

NOTE: Copying entry PORT.FORMATC from catalog ACATV6.FORMATS to catalog TEMP.FORMATS.
NOTE: PROCEDURE CATALOG used:
    real time 0.08 seconds
    cpu time 0.08 seconds

164  /* Create a binary transport file,*/
165  and populate it with the formats to be shared. */
166  file name TRANS 'C:\TEMP\Transport.bin';
167  proc cport lib = TEMP
168  file = TRANS;
169  run;

NOTE: Proc CPORT begins to transport catalog TEMP.FORMATS
NOTE: The catalog has 54 entries and its maximum
    logical record length is 92.
NOTE: Entry DELAYF.FORMAT has been transported.
NOTE: Entry PORT.FORMATC has been transported.
NOTE: PROCEDURE CPORT used:
    real time 0.14 seconds
    cpu time 0.01 seconds

172  /* This is a dummy Remote Submit
    statement.*/
173  we use it here to identify that the
code is to be run on
174  the receiving platform.
175  If we could remotely submit code, then
rather than
176  using the FTP access method, we would
use the Upload procedure.
177  Assign a file reference to the file on
the client machine.
178  to demonstrate the process we will use
the localhost IP
179  connection which loops back to the same
machine.
180  The Host parameter should contain
either the IP address or Host name
181  for the source machine. */
182  * RSubmit;
183  file name TRANS Ptp
184  'C:\\Temp\Transport.Bin';
185  host = 'localhost';
186  rcmd = 'binary';
187  User = 'christiana' pass =
188  XXXXXXX;

189  run;
190  libname HOSTTEMP 'C:\Windows\Temp';
NOTE: Libref HOSTTEMP was successfully assigned as follows:
191  Engine: V8
192  Physical Name: C:\\Windows\\Temp
193  /* Now use that file reference in the
    CImport step to reference
194  the format catalog on the host
195  */
and import the entries. */

Proc CImport Library = HOSTTEMP
InFile = TRANS;
Run;

NOTE: Proc CIMPORT begins to create/update
catalog HOSTTEMP.FORMATS
NOTE: Entry DELAYF.FORMAT has been imported.
NOTE: Entry PORT.FORMATC has been imported.
NOTE: Total number of entries processed in
catalog HOSTTEMP.FORMATS: 54

NOTE: PROCEDURE CIMPORT used:
real time 0.07 seconds
cpu time 0.06 seconds

Proc Catalog Cat = HOSTTEMP.Formats;
Copy Out = WORK.FORMATS;
Quit;

NOTE: Copying entry DELAYF.FORMAT from catalog
HOSTTEMP.FORMATS to catalog WORK.FORMATS.
NOTE: Copying entry PORT.FORMATC from catalog
HOSTTEMP.FORMATS to catalog WORK.FORMATS.
NOTE: PROCEDURE CATALOG used:
real time 0.04 seconds
cpu time 0.04 seconds

If we share the formats using the SAS/Connect Product, then the
following code is all that is required.

RSubmit;
LibName HOSTTEMP 'C:\Windows\Temp';
Proc UpLoad InLib = LIBRARY
OutLib = HOSTTEMP;
Select FORMATS / MemType = CATALOG;
Run;
Proc Catalog Cat = HOSTTEMP.Formats;
Copy Out = WORK.FORMATS;
Quit;

Neither of these processes is going to work if the local catalog is
created under Version 8, and the remote system is running under
Version 6. The reason for that is that the catalog we are sharing
across the two environments was created in a format unreadable
on the host.

I think the easiest way to share formats where the SAS versions
are, or may be different, is to use a process we have discussed
already. We export the format catalog to a data set, transfer or
share the data set, and then read the formats into the target
catalog. Here is some code to demonstrate this process.

/* The temporary library is assigned with the
engine appropriate to the host system. Here, we
are assuming that the host is running Version 6,
so the data set we are creating is in a
structure which can be read by the host. */
Libname TEMP V6 'C:\Temp';
Proc Format Lib = LIBRARY
CntlOut = TEMP.CONTROL;
Run;
RSubmit;

Proc UpLoad InLib = TEMP
OutLib = WORK;
Select CONTROL;
Run;

One last caution however. Be aware that the data sorting
sequence may be different between the two systems. While
Windows and Unix use ASCII as the character collation
sequence, a mainframe will usually use EBCDIC. In ASCII,
numbers have the rank values of 48 to 57, upper case characters
start at 65, and lower case characters start at 97. On EBCDIC
systems, the numeric values have a higher rank. Consequently,
on an ASCII coded system, a format covering 0 – Z will be valid.
On EBCDIC it will not. Similarly, on one system a format from A0
– AZ will be valid, and on the other it will not. This is only an
issue if we create character formats that include numbers in the
value range, but it is one of which we should be aware.

CONCLUSION

We have:
• looked at processes that create character and numeric
formats,
• discussed the application of keywords including ‘High’,
‘Low’ and ‘Other’ to expand the power of these formats,
• explored the data produced from the Format
Procedure’s CNTLOUT option,
• built our own data set to load a format table
dynamically from a data source that may regularly
change,
• reviewed some of the methods for sharing formats
between programs, and users
• applied descriptive labels to our formats to assist other
users
• considered a number of methods of sharing formats
across platforms.

All the code used in this paper, and the code to generate the
sample data used, is available on the authors website through the
page http://www.dkvj.com/. Click on the navigation bar for
‘Presentations’ and select this paper under ‘SUGI 28’.

CONTACT INFORMATION

Your comments, suggestions and questions are valued and
encouraged. Please contact the author:
David Johnson
DKV-J Consultancies
Ci’- Bonds Cottage,
Holmeswood Rd
Holmeswood nr Rufford
Lancashire England L40 1UA
Business Phone: +44 (0)7005 98 0828
Fax: +44 (0)7092 25 9556
Email: sugi28@dkvj.com
Web: http://www.dkvj.com

© 2000-2003, drawn with SAS/Graph®

DKV-J Consultancies
Business Information Systems